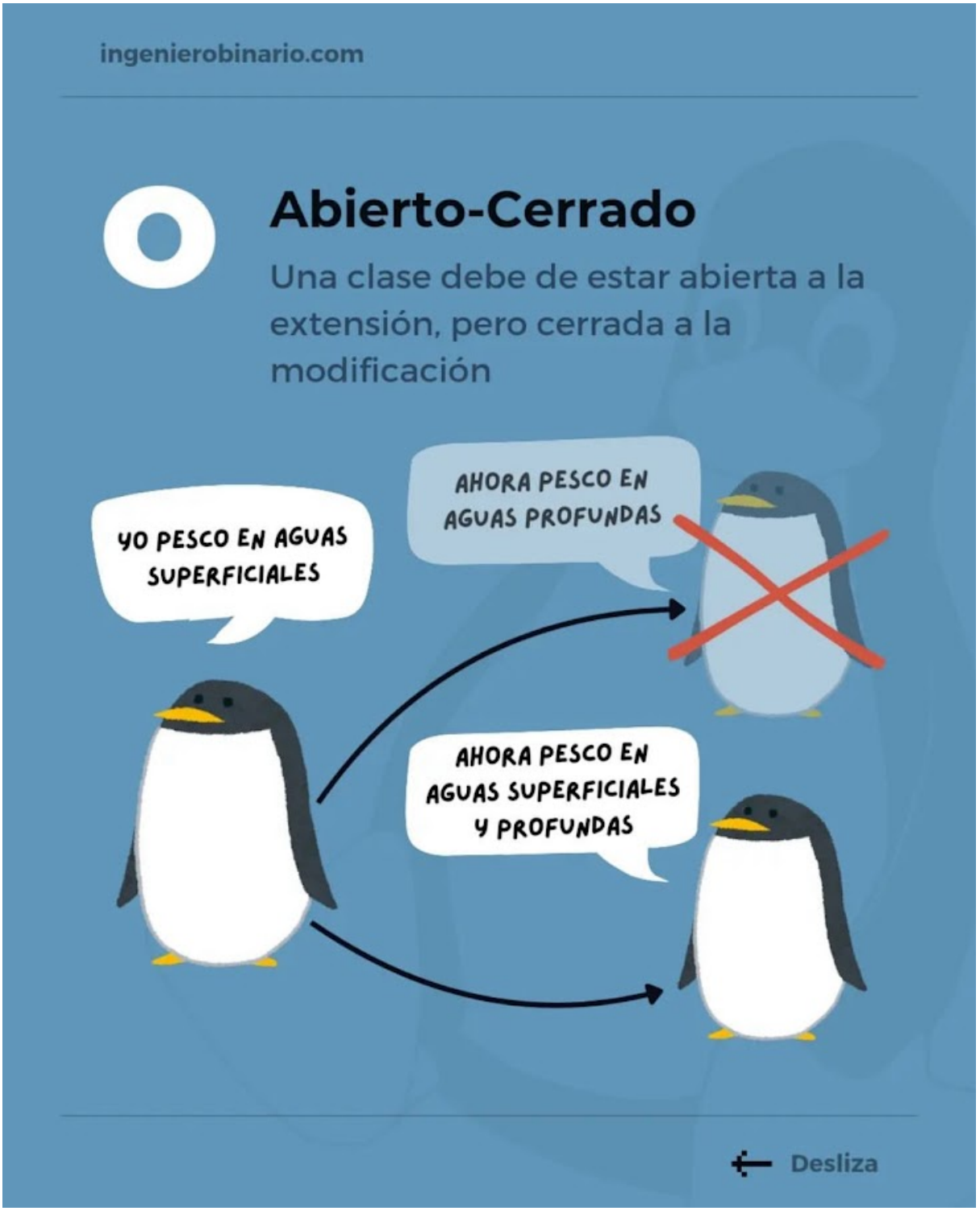
SOLID

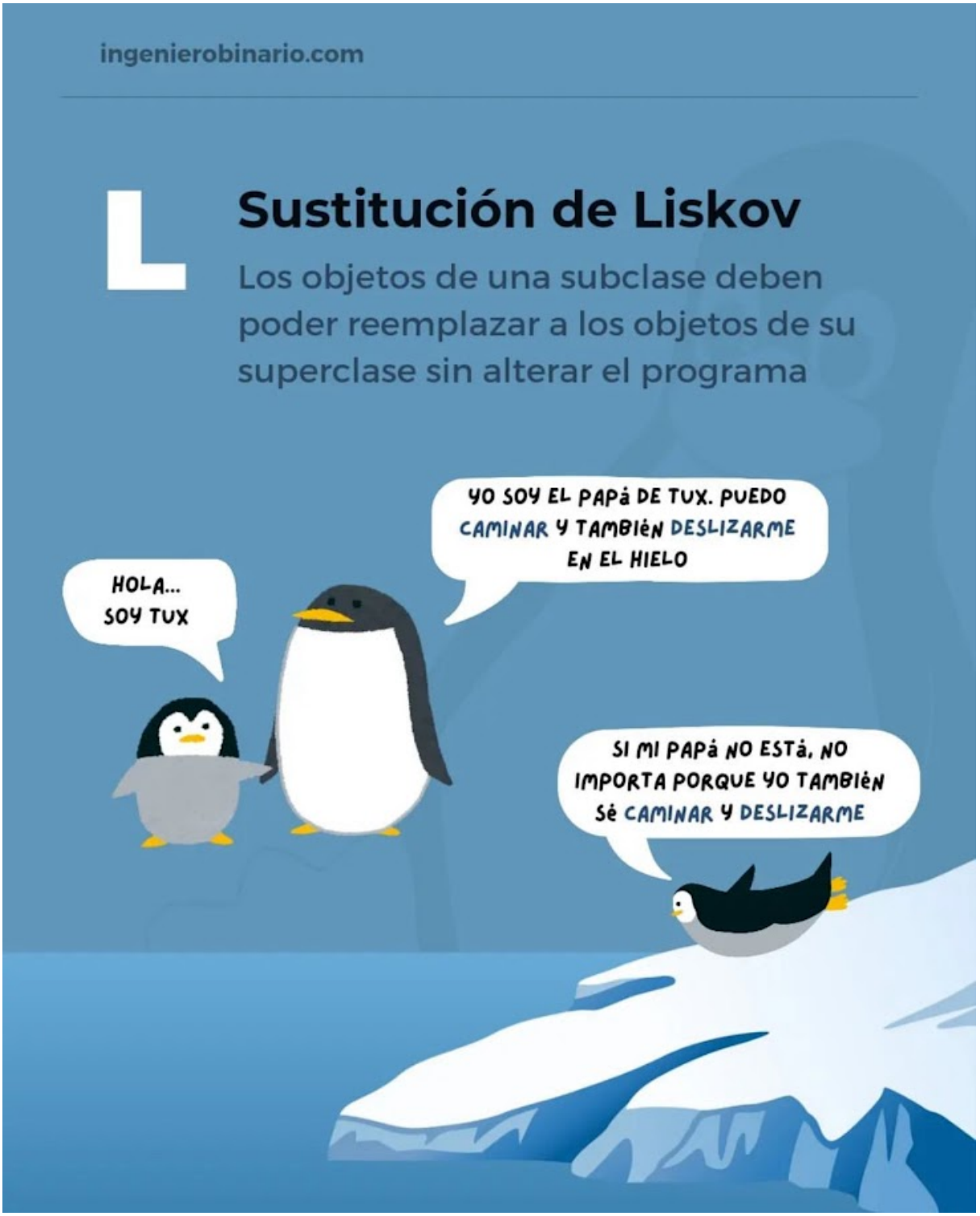

ingenierobinario.com
LOS PRINCIPIOS
SOLID
EXPLICADOS CON PINGüINOS
Desliza


ingenierobinario.com
SOLID son las siglas de los 5 principios más importantes para escribir código orientado a objetos, mantenible y escalable
Desliza


ingenierobinario.com
Generalmente se explican directamente con código, y eso está bien
Pero que te los expliquen con pinguinos...
YA ES MUCHO LEVEL ¿SÍ O QUÉ?
Desliza


ingenierobinario.com
S
Responsabilidad única
Cada clase debería tener una sola responsabilidad
YO PESCO PECES Y CONSTRUYO EL NIDO
YO SOLO PESCO PECES
Desliza


ingenierobinario.com
O
Abierto-Cerrado
Una clase debe de estar abierta a la extensión, pero cerrada a la modificación
YO PESCO EN AGUAS SUPERFICIALES
AHORA PESCO EN AGUAS PROFUNDAS
AHORA PESCO EN AGUAS SUPERFICIALES Y PROFUNDAS
Desliza


ingenierobinario.com
L
Sustitución de Liskov
Los objetos de una subclase deben poder reemplazar a los objetos de su superclase sin alterar el programa
HOLA... SOY TUX
YO SOY EL PAPá DE TUX. PUEDO CAMINAR Y TAMBIÉN DESLIZARME EN EL HIELO
SI MI PAPá NO ESTá, NO IMPORTA PORQUE YO TAMBIÉN SÉ CAMINAR Y DESLIZARME


ingenierobinario.com
I
Segregación de interfaces
Evita las interfaces grandes, pues pueden forzar la implementación de métodos innecesarios
TODOS LOS PINGUINOS DEBEN:
- CONSTRUIR UN NIDO
- PESCAR 10 PECES
- CUIDAR UN HUEVO
PERO SI SOLO SÉ PESCAR
SI A DURAS RESPIRO WE
SÉ TODO MENOS CONSTRUIR
Desliza


ingenierobinario.com
D
Inversión de dependencias
Los módulos de alto nivel no deben depender de los de bajo nivel, sino que ambos deben depender de abstracciones
PUEDO PESCAR SI ES CON UNA DE ESTAS REDES
YO SE UTILIZAR LA HERRAMIENTA QUE SEA
Desliza


SÍGUENOS PARA MÁS
CONTENIDO ASÍ
¿AHORA SÍ
LOS ENTENDISTE?
No vayas sin dejar tu ❤️ porfis
GUÁRDALO PARA MÁS TARDE

SOLID es un acrónimo de los **cinco principios del diseño orientado a objetos** (*OOD Object Oriented Design*) creados por Uncle Bob, quien también es coautor de los principios del desarrollo web agile.

Los cinco principios son:

- **S**. Single responsibility principle
- **O**. Open/Closed principle
- **L**. Liskov substitution principle
- **I**. Interface segregation principle
- **D**. Dependency Inversion Principle

Estos principios combinados facilitan al desarrollador crear proyectos fáciles de mantener y expandir.

1. Single responsibility principle

Una clase sólo debe tener un motivo para cambiar, lo que significa que sólo debe tener una tarea.

Tenemos varias figuras de las que después queremos calcular su área total:

```
Class Circle
{
    public $radius;

    public function __construct($radius)
    {
        $this->radius = $radius;
    }
}

Class Square
{
    public $length;

    public function __construct($length)
    {
        $this->length = $length;
```

```
    }
}
```

Primero creamos las clases de las figuras y dejamos que los constructores se encarguen de recibir las medidas necesarias.

Ahora creamos la clase **AreaCalculator**, que recibe un array con los objetos de cada una de las figuras para ser sumadas:

```
class AreaCalculator
{
    protected $shapes;

    public function __construct($shapes = array())
    {
        $this->shapes = $shapes;
    }

    public function sum()
    {
        // Aquí va la lógica para sumar todas las áreas
    }

    public function output()
    {
        return implode('', array(
            "<h1>",
                "Suma de todas las áreas: ",
                $this->sum(),
            "</h1>"
        ));
    }
}
```

Para utilizar la clase **AreaCalculator** simplemente instanciamos la clase y le pasamos un array con las figuras, mostrando el output al final:

```
$shapes = array (
    new Circle(3),
```

```
    new Square(4)
);

$areas = new AreaCalculator($shapes);

echo $areas->output();
```

El problema del método output es que la clase **AreaCalculator** además de calcular las áreas maneja la lógica de la salida de los datos. El problema surge cuando queremos mostrar los datos en otros formatos como json, por ejemplo.

El **principio Single responsibility** determinaría en este caso que AreaCalculator sólo calculase el área, y que la funcionalidad de la salida de los datos de produjera en otra entidad. Para ello podemos crear la clase SumCalculatorOutputter, que determinará como mostraremos los datos de las figuras. Con esta clase el código quedaría así:

```
$shapes = array (
    new Circle(3),
    new Square(4)
);

$areas = new AreaCalculator($shapes);
$output = new SumCalculatorOutputter($areas);

echo $output->toJson();
echo $output->toHtml();
```

2. Open/Closed principle

Los objetos o entidades deberían estar abiertas a su extensión, pero cerradas para su modificación.

Este principio quiere decir que una clase debería ser fácilmente extendible sin modificar la propia clase. Vamos a ver ahora el método *sum* de la clase **AreaCalculator**:

```
public function sum()
{
    foreach ($this->shapes as $shape) {
```

```
        if(is_a($shape, 'Square')){
            $area[] = pow($shape->length, 2);
        } elseif (is_a($shape, 'Circle')){
            $area[] = pi() * pow($shape->radius, 2);
        }
    }
    return array_sum($area);
}
```

Si quisiéramos que el métdo *sum* pudiera calcular la suma de más figuras, tendríamos que seguir añadiendo bloques if/else, lo que va en contra del principio Open/Closed.

Una forma de hacer este método *sum* mejor es moviendo la lógica de calcular el area a la clase de cada figura, añadiendo un método area() en cada clase:

```
Class Square
{
// ...
    public function area()
    {
        return pow($this->length, 2);
    }
}
```

Lo mismo se hará en la clase Circle:

```
Class Circle
{
// ...
    public function area()
    {
        return pi() * pow($this->radius, 2);
    }
}
```

Ahora para calcular la suma de las figuras proporcionadas dejaremos el método *sum* de la siguiente forma:

```
public function sum()
{
    foreach ($this->shapes as $shape)
    {
        $area[] = $shape->area;
    }

    return array_sum($area);
}
```

Ahora podemos crear cualquier otra figura y pasarla para calcular la suma que no se romperá el código. Ahora la pregunta es la siguiente: ¿Cómo sabemos que el objeto que se pasa a **AreaCalculator** es realmente una figura o si la figura tiene un método llamado *área*?

Crear interfaces es una parte integral de los **principios SOLID**. Vamos a crear una interface que ha de implementar cada figura:

```
interface ShapeInterface {
    public function area();
}
```

Ahora todas las figuras deberán implementarla:

```
Class Circle implements ShapeInterface
{
// ...
}
Class Square implements ShapeInterface
{
// ...
}
```

En el método *sum* de **AreaCalculator** podemos comprobar si las figuras proporcionadas son realmente instancias de **ShapeInterface**, y sino, lanzar una excepción:

```
public function sum()
{
    foreach ($this->shapes as $shape) {
        if($shape instanceof ShapeInterface)){
            $area[] = $shape->area;
            continue;
        }
        throw new AreaCalculatorInvalidShapeException;
    }

    return array_sum($area);
}
```

3. Liskov substitution principle

Si S es una subclase de T, entonces los objetos de T podrían ser substituidos por objetos del tipo S sin alterar las propiedades del problema. Esto es, cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

Lo que quiere decir es que cualquier subclase debería poder ser sustituible por la clase padre.

Continuando con la clase **AreaCalculator**, ahora tenemos una clase **VolumeCalculator** que extiende la clase AreaCalculator:

```
class VolumeCalculator extends AreaCalculator
{
    public function __construct($shapes = array())
    {
        parent::__construct($shapes);
    }

    public function sum()
    {
        // Calcula el volumen y devuelve un array de salida
        $summedData = '';
        return $summedData;
```

```
    }
}
```

VolumeCalculator se podría sustituir por AreaCalculator.

La clase **SumCalculatorOutputter** quedará:

```
class SumCalculatorOutputter {

    protected $calculator;

    public function __construct(AreaCalculator $calculator)
    {
        $this->calculator = $calculator;
    }

    public function toJson()
    {
        $data = array (
          'sum' => $this->calculator->sum()
        );

        return json_encode($data);
    }

    public function toHtml()
    {
        return implode('', array(
            '<h1>',
                'Suma de las áreas de las figuras: ',
                $this->calculator->sum(),
            '</h1>'
        ));
    }
}
```

4. Interface segregation principle

Una clase nunca debe ser forzada a implementar una interface que no usa, empleando métodos que no tiene por qué usar.

De nuevo en el ejemplo de figuras, sabemos que también tenemos figuras con volumen, por lo que podríamos añadir el método *volume* en la interface **ShapeInterface**:

```
interface ShapeInterface {
    public function area();
    public function volume();
}
```

Cualquier figura que creemos debe implementar el método *volume*, pero esto no es lo que queremos ya que fuerza a las figuras sin volumen a aplicar este método. Para solucionarlo podríamos crear una nueva interface **SolidShapeInterface**:

```
interface ShapeInterface
{
    public function area();
}

interface SolidShapeInterface
{
    public function volume();
}

class Cube implements ShapeInterface, SolidShapeInterface
{
    public function area()
    {
        // Calcula la superficie del cubo
    }

    public function volume()
    {
        // Calcula el volumen del cubo
```

```
    }
}
```

Esta forma es mejor, pero a la hora de hacer type hinting habría que elegir entre ShapeInterface o SolidShapeInterface.

Para solucionar esto, podemos crear otra interface, **ManageShapeInterface**, e implementarla en las figuras con y sin volumen, de esta forma puedes ver fácilmente que tiene un API para administrar las figuras:

```
interface ManageShapeInterface {
    public function calculate();
}

class Square implements ShapeInterface, ManageShapeInterfac
e {
    public function area() { /*Hacer cálculos*/ }

    public function calculate() {
        return $this->area();
    }
}

class Cube implements ShapeInterface, SolidShapeInterface,
ManageShapeInterface {
    public function area() { /*Hacer cálculos*/ }
    public function volume() { /*Hacer cálculos*/ }

    public function calculate() {
        return $this->area() + $this->volume();
    }
}
```

Ahora en la clase **AreaCalculator** podemos reemplazar la llamada al método *area* por *calculate* y comprobar si el objeto es una instancia de **ManageShapeInterface** y no de **ShapeInterface**.

5. Dependency inversion principle

Las entidades deben depender de abstracciones no de concreciones. El módulo de alto nivel no debe depender del módulo de bajo nivel, pero deben depender de abstracciones.

Cambiamos ahora el ejemplo por uno relacionado con **bases de datos**:

```
class PasswordReminder
{
    private $dbConnection;

    public function __construct(MySQLConnection $dbConnecti
on)
    {
        $this->dbConnection = $dbConnection;
    }
}
```

MySQLConnection es el módulo de bajo nivel, mientras que **PasswordReminder** es de alto nivel. Este ejemplo no respeta el **principio SOLID** de **dependency inversion** ya que se está forzando a la clase PasswordReminder a depender en la clase MySQLConnection.

Si después quieres cambiar el motor de base de datos tendrás que cambiar la clase PasswordReminder también, lo que viola el **principio open-closed**.

A la clase PasswordReminder no debería importarle que base de datos emplea tu **aplicación**, y para solucionarlo empleamos una interface:

```
interface DBConnectionInterface
{
    public function connect();
}
```

La interface tiene un método connect y la clase MySQLConnection implementa esta interface. En lugar de hacer type hinting con la clase MySQLConnection en PasswordReminder, lo hacemos con la interface, de forma que no importa el tipo de base de datos que empleemos, que PasswordReminder conectará a la base de datos sin problemas:

```
class MySQLConnection implements DBConnectionInterface {
    public function connect() {
        return "Conexión a la base de datos";
    }
}

class PasswordReminder {
    private $dbConnection;

    public function __construct(DBConnectionInterface $dbCo
nnection) {
        $this->dbConnection = $dbConnection;
    }
}
```

Ahora podemos ver que tanto los niveles altos como los bajos dependen de abstracciones.

Fuentes: scotch.io